On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers

Patrick Amestoy, Jean-Yves L'Excellent, Gilles Moreau

On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers

Patrick Amestoy, Jean-Yves L'Excellent, Gilles Moreau

RESEARCH REPORT
N° 9122
Décembre 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9122--FR+ENG

On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers

Patrick Amestoy[*], Jean-Yves L'Excellent[†], Gilles Moreau[†]

Project-Team ROMA

Research Report n° 9122 — Décembre 2017 — 28 pages

Abstract: The cost of the solution phase in sparse direct methods is sometimes critical. It can be larger than the one of the factorization in applications where systems of linear equations with thousands of right-hand sides (RHS) must be solved. In this paper, we focus on the case of multiple *sparse* RHS with different nonzero structures in each column. Given a factorization $A = LU$ of a sparse matrix A and the system $AX = B$ (or $LY = B$ when focusing on the forward elimination), the sparsity of B can be exploited in two ways. First, *vertical* sparsity is exploited by pruning unnecessary nodes from the elimination tree, which represents the dependencies between computations in a direct method. Second, we explain how *horizontal* sparsity can be exploited by working on a subset of RHS columns at each node of the tree. A combinatorial problem must then be solved in order to permute the columns of B and minimize the number of operations. We propose a new algorithm to build such a permutation, based on the tree and on the sparsity structure of B. We then propose an original approach to split the columns of B into a minimal number of blocks (to preserve flexibility in the implementation or maintain high arithmetic intensity, for example), while reducing the number of operations down to a given threshold. Both algorithms are motivated by geometric intuitions and designed using an algebraic approach, and they can be applied to general systems of linear equations. We demonstrate the effectiveness of our algorithms on systems coming from real applications and compare them to other standard approaches. Finally, we give some perspectives and possible applications for this work.

Key-words: sparse linear algebra, sparse matrices, direct method, multiple sparse right-hand sides

[*] University of Toulouse, INPT and IRIT laboratory, France
[†] University of Lyon, Inria and LIP laboratory, France

Exploitation de seconds-membres creux et multiples dans les solveurs creux directs

Résumé : Le coût des résolutions triangulaires des solveurs creux directs est parfois critique. Ce coût peut dépasser celui de la factorisation dans les applications qui nécessitent la résolution de plusieurs milliers de seconds membres. Cette étude se concentre sur les cas où les seconds membres sont multiples, creux et n'ont pas tous la même structure.

Étant donnée la factorisation $A = LU$ d'une matrice creuse A, l'étude met surtout l'accent sur la résolution du premier système triangulaire $LY = B$, où L est triangulaire inférieure. Dans ce type de problèmes, le creux dans la matrice B peut être exploité de deux manières. Premièrement, on évite le calcul de certaines lignes, ce qui correspond à élaguer certains nœuds de l'arbre d'élimination (qui représente les dépendances entre les calculs de la résolution). Deuxièmement, on réduit, pour chaque nœud, le calcul sur des sous-ensembles de colonnes de B plutôt que sur la matrice complète. Dans ce cas, un problème combinatoire doit être résolu afin de trouver une permutation des colonnes de B.

S'appuyant d'abord sur l'algorithme de dissection emboitée appliqué à un domaine régulier, un premier algorithme est proposé pour contruire une permutation des colonnes de B. Puis une nouvelle approche permet de poursuivre la réduction du nombre d'opérations grâce à la création de blocs. Pour préserver la flexibilité de l'implémentation ainsi que l'efficacité des opérations de type BLAS 3, un nombre minimal de groupe est créé. Inspirés d'abord par des observations géométriques, ces nouveaux algorithmes ont été étendus algébriquement pour n'utiliser que des informations provenant de la structure des seconds membres et des arbres d'élimination. Ils permettent ainsi une convergence rapide vers le nombre minimal d'opérations. Les résultats expérimentaux démontrent le gain obtenu par rapport à d'autres approches classiques. Enfin, les applications et extensions possibles de ce travail sont présentées.

Mots-clés : Algèbre linéaire creuse, matrices creuses, méthode directe, seconds membres creux et multiples

Introduction

We consider the direct solution of sparse systems of linear equations

$$AX = B, \tag{1}$$

where A is an $n \times n$ sparse matrix with a symmetric structure and B is an $n \times m$ matrix of right-hand sides (RHS). When A is decomposed under the form $A = LU$ with a sparse direct method [7], *e.g.*, the multifrontal method [8], the solution can be obtained by forward and backward triangular solves involving L and U. In this study, we are interested in the situation where not only A is sparse, but also B, with the columns of B possibly having different structures, and we focus on the efficient solution of the forward system

$$LY = B, \tag{2}$$

where the unknown Y and the right-hand side B are $n \times m$ matrices. We will see in this study that the ideas developed for Equation (2) are indeed more general and can be applied in a broader context. In particular, they can be applied to the backward substitution phase, in situations where the system $UX = Y$ must be solved for a subset of the entries in X [3, 17, 19, 20]. In direct methods, the dependencies of the computations for factorization and solve operations can be represented by a tree [13], which plays a central role and expresses parallelism between tasks. The factorization phase is usually the most costly phase but, depending on the number of columns m in B or on the number of systems to solve with identical A and different B, the cost of the solve phase may also be significant. As an example, electromagnetism, geophysics or imaging applications can lead to systems with sparse multiple right-hand sides for which the solution phase is significantly more costly than the factorization phase [1, 16]. Such applications motivate the algorithms presented in this study.

A sparse RHS is characterized by its set of nonzeros and it is worth considering a RHS as sparse when doing so improves performance or storage compared to the dense case. The exploitation of RHS sparsity (later extended to reduce computations when only a subset of the solution is needed [17, 19]) was formalised by Gilbert [10] and Gilbert and Liu [11], who showed that the structure of the solution Y from Equation (2) can be predicted from the structures of L and B. From this structure prediction, one can design mechanisms to reduce computation. In particular, *tree pruning* suppresses nodes in the elimination tree involving only computations on zeros. When solving a problem with multiple RHS, the preferred technique is usually to process all the RHS in one shot. The subset of the elimination tree to be traversed is then the union of the different pruned trees (see Section 2.1). However, when the RHS have different structures, this means that extra operations on zeros have to be performed. In order to limit these extra operations, several approaches may be applied. The one that minimizes the number of operations consists in processing the RHS columns one by one, each time with a different pruned tree. However, such an approach is not practical and leads to a poor arithmetic intensity (*e.g.*, it will not exploit level 3 BLAS [6]). Another approach, in the context of blocks of RHS with a predetermined number of columns in each block, consists in applying heuristics to determine which columns to include within which block. The objective function to minimize might be the volume of accesses to the factor matrices [3], or the number of operations [18]. When possible, large sets of columns, possibly the whole set of m columns, may be processed in one shot. Thanks to the different sparsity structure of each column of B, it is then possible to work on less than m columns at most nodes in the tree, as explained in Section 2.2. Such a mechanism has been introduced in the context of the parallel computation of entries of the inverse [4], where at each node, computations are performed on a contiguous interval of RHS columns.

After a description of these mechanisms with illustrative examples, one contribution of this work is to propose algorithms that improve the exploitation of *column intervals* at each node. A geometrical intuition motivates a new approach to obtain a permutation of the columns of B that significantly reduces computation during (2) with respect to previous work. The algorithm is first introduced for a nested dissection ordering and for a regular mesh, then generalized to arbitrary elimination trees. Computation can then be further reduced by dividing the RHS into blocks. However, instead of enforcing a constant number of columns per block, our objective is to minimize the number of blocks created. If $\Delta_{min}(B)$ represents the number of operations to solve (2) when processing the RHS columns one by one, we show on real applications that our blocking algorithm can approach $\Delta_{min}(B)$ within a tolerance of 1% while creating a small number of blocks. Please note that RHS sparsity limits the amount of tree parallelism because only a few branches are traversed in the elimination tree. Therefore, whenever possible, our heuristics also aim at choosing the approach that maximizes tree parallelism.

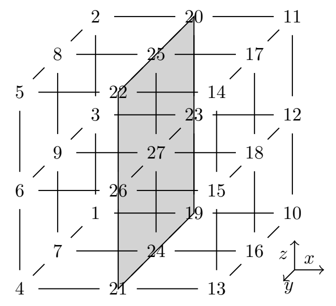
This paper is organized as follows. Section 1 presents the general context of our study and Section 2 exposes the classical *tree pruning* technique together with the notion of *node intervals* where different intervals of columns may be processed at each node of the tree. In Section 3, we introduce a new permutation to reduce the size of such intervals and thus limit the number of operations, first using geometrical considerations for a regular nested dissection ordering, then with a pure algebraic approach that can then be applied in a general case and for arbitrary right-hand sides. We call it the *Flat Tree* algorithm because of the analogy with the ordering that one would obtain when "flattening" the tree. In Section 4, an original blocking algorithm is then introduced to further improve the flat tree ordering. It aims at defining a limited number of blocks of right-hand sides to minimize the number of operations while preserving parallelism. Section 5 gives experimental results on a set of systems coming from two geophysics applications relying on Helmholtz or Maxwell equations. Section 6 discusses adaptations of the nested dissection algorithm to further decrease computation and Section 7 shows why this work has a broader scope than solving Equation (2) and presents possible applications.

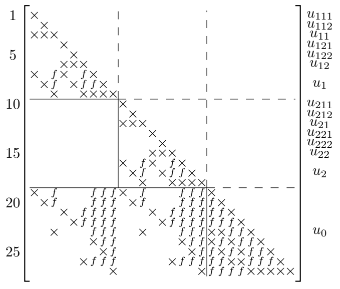
1 Nested dissection, sparse direct solvers and triangular solve

In sparse direct methods, the order of the variables of a sparse matrix A strongly impacts the number of operations for the factorization, the size of the factor matrices, and the cost of the solve phase. We illustrate in Figure 1 the use of nested dissection on a regular mesh [9].

The nested dissection algorithm consists in dividing domains with separators. The regular $3\times3\times3$ domain shown in Figure 1(a) is first divided by a 3×3 constant-x plane separator (variables $\{19,\ldots,27\}$ forming separator u_0) into two even subdomains. Each subdomain is then divided recursively (constant-y plane separators $\{16,17,18\}$ and $\{7,8,9\}$, etc.). By ordering the separators after the subdomains, large blocks of zeros limit the amount of computation (Figure 1(b)). Although it could be different, the order inside each separator is similar to [9, Appendix].

The elimination tree [13] represents dependencies between computations: it is processed from bottom to top in the factorization and forward elimination, and from top to bottom in the backward substitution. The elimination tree may be compressed thanks to the use of supernodes, leading to a tree identical to the *separator tree* of Figure 1(c) when choosing supernodes identical to the sepa-


(a) $3 \times 3 \times 3$ regular mesh.


(b) Structures of A and L.

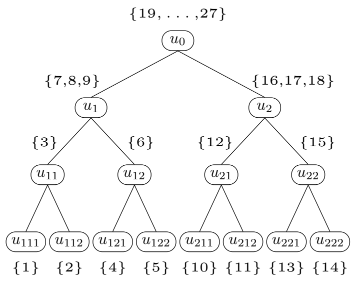

(c) Separator tree T.

Figure 1: (a) A 3D mesh with a 7-point stencil. Mesh nodes are numbered according to a nested dissection ordering. (b) Corresponding matrix with initial nonzeros ($\times$) in the lower triangular part of a symmetric matrix A and fill-in (f) in the L factor. (c) Separator tree, also showing the sets of variables to be eliminated at each node.

rators resulting from the nested dissection algorithm[1]. We note that the order in which tree nodes are processed (u_{111}, u_{112}, u_{11}, ..., u_0), represented on the right of the matrix, is a postordering: nodes in any subtree are processed consecutively.

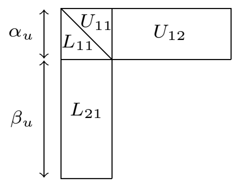


Figure 2: Structure of the factors associated to a node u of the tree.

Considering a single RHS b and the decomposition $A = LU$, the solution of the triangular system $Ly = b$ (and $Ux = y$) relies on block operations at each node of the tree T. Figure 2 represents the L and U factors restricted to a given node u of T, where the diagonal block is formed of the two lower and upper triangular matrices L_{11} and U_{11}, and the *update* matrices are L_{21} and U_{12}. The α_u variables are the ones of node (or separator) u, and the β_u variables correspond to the nonzero rows in the off-diagonal parts of the L factor restricted to node u (Figure 1(b)), that have been gathered together. For example, node u_1 from Figure 1 corresponds to separator $\{7, 8, 9\}$, so that L_{11} and U_{11} are of order $\alpha_{u_1} = 3$ and there are $\beta_{u_1} = 9$ update variables $\{19, \ldots, 27\}$, so that L_{21} is of size 9×3 (and U_{12} is of size 3×9). Starting with $y \leftarrow b$, the active components of y are gathered into two temporary dense vectors y_1 of size α_u and y_2 of size β_u at each node u of T, where the triangular solve

$$y_1 \leftarrow L_{11}^{-1} y_1, \tag{3}$$

[1]Note that in this example, identifying supernodes to separators leads to relaxed supernodes: although some sparsity exists in the interaction of u_7 and u_0 (and u_{112} and u_0), the interaction is considered dense and few computations on zeros are performed to benefit from larger blocks.

is performed, followed by the update operation

$$y_2 \leftarrow y_2 - L_{21} y_1. \tag{4}$$

y_1 and y_2 can then be scattered back into y, and y_2 will be used at higher levels of T. When the root is processed, y contains the solution of $Ly = b$. Because the matrix blocks in Figure 2 are considered dense, there are $\alpha_u(\alpha_u - 1)$ arithmetic operations for the triangular solution (3) and $2\alpha_u\beta_u$ operations for the update operation (4), leading to a total number of operations

$$\Delta = \sum_{u \in T} \delta_u, \tag{5}$$

where $\delta_u = \alpha_u \times (\alpha_u - 1 + 2\beta_u)$ is the number of arithmetic operations at node u.

2 Exploitation of sparsity in right-hand sides

In this section, we review two approaches to exploit sparsity in B when solving the triangular system (2). The first one, called tree pruning [11, 17] and explained in Section 2.1, consists in pruning the nodes at which only computations on zeros are performed. The second one, presented in Section 2.2, goes further by working on different sets of RHS columns at each node of the tree [4].

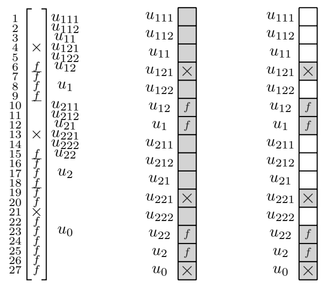
2.1 Tree pruning

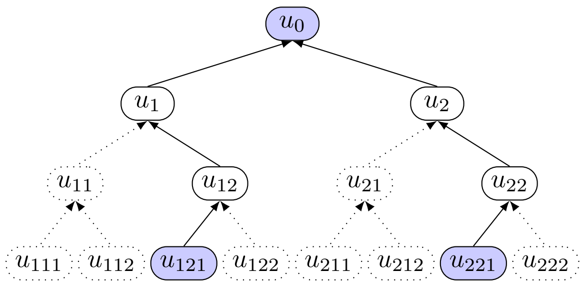
Consider a non-singular $n \times n$ matrix A with a nonzero diagonal, and its directed graph $G(A)$, with an edge from vertex i to vertex j if $a_{ij} \neq 0$. Given a vector b, let us define $\text{struct}(b) = \{i, b_i \neq 0\}$ as its nonzero structure, and $\text{closure}_A(b)$ as the smallest subset of vertices of $G(A)$ including $\text{struct}(b)$ without incoming edges. Gilbert [10, Theorem 5.1] characterizes the structure of the solution of $Ax = b$ by the relation $\text{struct}(A^{-1}b) \subseteq \text{closure}_A(b)$, with equality in case there is no numerical cancellation. In our context of triangular systems, ignoring such cancellation, $\text{struct}(L^{-1}b) = \text{closure}_L(b)$ is also the set of vertices reachable from $\text{struct}(b)$ in $G(L^T)$, where edges have been reversed [11, Theorem 2.1]. Finding these reachable vertices can be done using the transitive reduction of $G(L^T)$, which is a tree (the elimination tree) when L results from the factorization of a matrix with symmetric (or symmetrized) structure.

Since we work with a tree T with possibly more than one variable to eliminate at each node, let us define V_b as the set of nodes in T including at least one element of $\text{struct}(b)$. The structure of $L^{-1}b$ can be obtained by following paths from the nodes of V_b upto the root and these will be the only nodes needed to compute $L^{-1}b$. The tree consisting of this subset of nodes is what we call the *pruned tree* for b, and we note it $T_p(b)$. Thanks to this pruned tree, the number of operations Δ from Equation (5) now depends on b:

$$\Delta(b) = \sum_{u \in T_p(b)} \delta_u. \tag{6}$$

Example 2.1. Let b be a vector with nonzeros at positions 4, 13, and 21. The corresponding tree nodes are given by $V_b = \{u_{121}, u_{221}, u_0\}$, see Figures 1 and 3. Following the paths in T from nodes in V_b to the root results in the pruned tree of Figure 3(b). Compared to $\Delta = 288$ in the case of a dense right-hand side, $\Delta(b) = 228$ ($\delta_{u_{121}} = \delta_{u_{221}} = 6, \delta_{u_{12}} = \delta_{u_{22}} = 12, \delta_{u_2} = \delta_{u_1} = 60, \delta_{u_0} = 72$).


(a) Structure of $L^{-1}b$.



(b) Pruned tree $T_p(b)$.

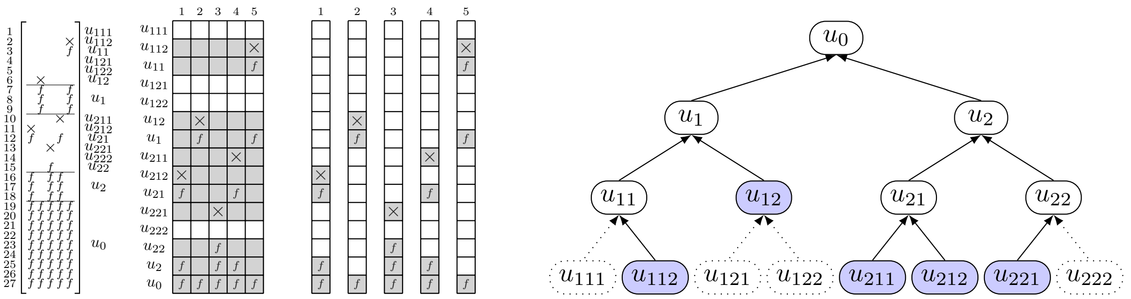
Figure 3: Illustration of Example 2.1. (a) Structure of $L^{-1}b$ with respect to matrix variables (left) and to tree nodes (middle and right). $\times$ corresponds to original nonzeros and f to fill-in. In the dense case (middle) and the sparse case (right), gray parts of $L^{-1}b$ are the ones involving computation. (b) Pruned tree $T_p(b)$: pruned nodes and edges are represented with dotted lines and nodes in V_b are filled.

We now consider the case of multiple RHS (Equation (2)), where RHS columns may have different structures and denote by B_i the columns of B, for $1 \leq i \leq m$. Instead of solving m linear systems with each pruned tree $T_p(B_i)$, one generally prefers to favor matrix-matrix computations for performance reasons. For that, a first approach consists in considering $V_B = \bigcup_{1\leq i\leq m} V_{B_i}$, the union of all nodes in T with at least one nonzero from matrix B, and the pruned tree $T_p(B) = \bigcup_{1\leq i\leq m} T_p(B_i)$ containing all nodes in T reachable from nodes in V_B. In that case, triangular and update operations (3) and (4) become $Y_1 \leftarrow L_{11}^{-1}Y_1$ and $Y_2 \leftarrow Y_2 - L_{21}Y_1$, at each node of the tree. The number of operations can then be defined as:

$$\Delta(B) = m \times \sum_{u\in T_p(B)} \delta_u. \tag{7}$$

Example 2.2. Figure 4(a) shows a RHS matrix $B = [\{B_{11,1}\}, \{B_{6,2}\}, \{B_{13,3}\},\{B_{10,4}\}, \{B_{2,5}\}]$ in terms of original variables (1 to 27) and in terms of tree nodes ($V_B = \{u_{212}, u_{12}, u_{221}, u_{211}, u_{112}\}$). In Figure 4(a), $\times$ corresponds to an initial nonzero in B and f corresponds to "fill-in" that appears in $L^{-1}B$ during the forward elimination on the nodes that are on the paths from nodes in V_B to the root (see Figure 4(b)). We have $\Delta(B) = 5 \times 264 = 1320$ and $\Delta(B_1) + \Delta(B_2) + \ldots + \Delta(B_5) = 744$.

At this point, we exploit tree pruning but perform extra operations by considering globally $T_p(B)$ instead of each individual pruned tree $T_p(B_i)$. In other words, we only exploit the *vertical sparsity* of B. Processing B by smaller blocks of columns would further reduce the number of operations at the cost of more traversals of the tree and a smaller arithmetic intensity, with a minimal number of operations $\Delta_{min}(B) = \sum_{i=1,m} \Delta(B_i)$ reached when B is processed column by column, as in Figure 4(a)(right). We note that performing this minimal number of operations while traversing the tree only once (and thus accessing the L factor only once) from leaves to root would require performing complex and costly data manipulations at each node u with copies and indirect accesses to work only on the active columns of B at u. We present in the next section a simpler approach which consists in exploiting the notion of intervals of columns at each node $u \in T_p(B)$. This approach to exploit what we call *horizontal sparsity* in B was introduced in another context [4].



(a) Structures of $L^{-1}B$, $L^{-1}B_1$, ..., $L^{-1}B_5$. (b) Pruned tree $T_p(B) = T_p(B_1) \cup \ldots \cup T_p(B_5)$.

Figure 4: Illustration of multiple RHS and tree pruning corresponding to Example 2.2. Gray parts of $L^{-1}B$ (resp. of $L^{-1}B_i$) are the ones involving computations when RHS are processed in one shot (resp. one by one).

2.2 Working with column intervals at each node

Given a matrix B, we associate to a node $u \in T_p(B)$ its set of active columns

$$Z_u = \{j \in \{1, \ldots, m\} \mid u \in T_p(B_j)\}. \tag{8}$$

The interval $[\![\min(Z_u), \max(Z_u)]\!]$ includes all active columns, and its length is

$$\theta(Z_u) = \max(Z_u) - \min(Z_u) + 1.$$

Z_u is sometimes defined for an ordered or partially ordered subset R of the columns of B, in which case we use the notations $Z_u|_R$, and $\theta(Z_u|_R)$. For u in $T_p(B)$, Z_u is non-empty and $\theta(Z_u)$ is different from 0. The main idea is then to perform the operations (3) and (4) on the $\theta(Z_u)$ contiguous columns $[\![\min(Z_u), \max(Z_u)]\!]$ instead of the m columns of B, leading to

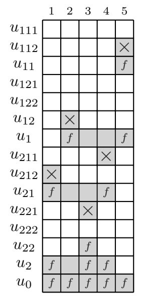
$$\Delta(B) = \sum_{u \in T_p(B)} \delta_u \times \theta(Z_u). \tag{9}$$

Example 2.3. In Example 2.2, there are nonzeros in columns 1 and 4 at node u_{21} so that $Z_{u_{21}} = \{1, 4\}$ (see Figure 5). Instead of performing the solve operations on all 5 columns at node u_{21}, we limit the computations to the $\theta(Z_{u_{21}}) = 4$ columns of interval $[\![1, 4]\!]$ (and to a single column at, *e.g.*, node u_{221}). Overall, $\Delta(B)$ is reduced from 1320 to 948 (while $\Delta_{min}(B) = 744$).

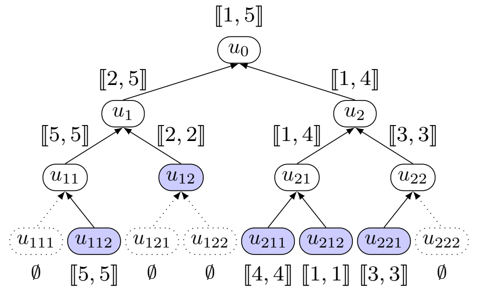
It is clear from Example 2.3 that $\theta(Z_u)$ and $\Delta(B)$ strongly depend on the order of the columns in B. In Section 3, we formalize the problem of permuting the columns of B and propose a new heuristic to find such a permutation. In Section 4, we further decrease the number of operations by identifying and extracting "problematic" columns.

3 Permuting RHS columns

We showed in Section 2.2 that *horizontal sparsity* can be exploited thanks to column intervals. The number of operations to solve (2) then depends on the permutation of the columns of B and we



(a) Structure of $L^{-1}B$.



(b) Pruned tree $T_p(B)$ with intervals.

Figure 5: Column intervals corresponding to Example 2.3: in gray (a) and above/below each node (b).

express the corresponding minimization problem as:

$$\begin{aligned}&\text{Find a permutation } \sigma \text{ of } \{1,\dots,m\} \text{ that minimizes } \Delta(B,\sigma) = \textstyle\sum_{u\in T_p(B)} \delta_u \times \theta(\sigma(Z_u)),\\ &\text{where } \sigma(Z_u) = \{\sigma(i) \mid i \in Z_u\}\,, \text{ and} \qquad (10)\\ &\qquad \theta(\sigma(Z_u)) \text{ is the length of the permuted interval } [\![\min(\sigma(Z_u)), \max(\sigma(Z_u))]\!].\end{aligned}$$

Rather than trying to solve the global problem with linear optimization techniques, we will propose a cheap heuristic based on the tree structure. We first define the notion of node optimality.

Definition 3.1. Given a node u in $T_p(B)$, and a permutation σ of $\{1,\dots,m\}$, we say that we have node optimality at u, or that σ is $u-$optimal if and only if $\theta(\sigma(Z_u)) = \#Z_u$, where $\#Z_u$ is the cardinal of Z_u. Said differently, $\sigma(Z_u)$ is a set of contiguous elements.

Remark that $\theta(\sigma(Z_u)) - \#Z_u$ is the number of extra columns (or padded zeros) on which extra computation is performed and is equal to 0 in case σ is $u-$optimal.

Example 3.1. Consider the RHS structure of Figure 5(a) and the identity permutation. We have node optimality at u_0 because $\#Z_{u_0} = \#\{1,2,3,4,5\} = 5 = \theta(Z_{u_0})$. We do not have node optimality at u_1 and u_2 because the numbers of padded zeros are $\theta(Z_{u_1}) - \#Z_{u_1} = 2$ and $\theta(Z_{u_2}) - \#Z_{u_2} = 1$, respectively. Our aim is thus to find a permutation σ that reduces the difference $\theta(\sigma(Z_u)) - \#Z_u$.

In the following, we first present a permutation based on a postordering of $T_p(B)$, then expose our new heuristic, which targets node optimality in priority at the nodes near the top of the tree.

3.1 The Postorder permutation

In Figure 1, the sequence [u_{111}, u_{112}, u_{11}, u_{121}, u_{122}, u_{12}, u_1, u_{211}, u_{212}, u_{21}, u_{221}, u_{222}, u_{22}, u_2, u_0] used to order the matrix follows a postordering: any subtree contains a set of consecutive nodes in that sequence. This postordering is also the basis to permute the columns of B:

Definition 3.2. Consider a postordering of the tree nodes $u \in T$, and a RHS matrix $B = [B_j]_{j=1\dots m}$ where each column B_j is represented by one of its associated nodes $u(B_j) \in V_{B_j}$ (see below). B is said to be postordered if and only if: $\forall j_1, j_2, 1 \leq j_1 < j_2 \leq m$, we have either $u(B_{j_1}) = u(B_{j_2})$, or $u(B_{j_1})$ appears before $u(B_{j_2})$ in the postordering. In other words, the order of the columns B_j is compatible with the order of their representative nodes $u(B_j)$.

The postordering has been applied [3, 17, 18] to group together in regular chunks RHS columns with "nearby" pruned trees, thereby limiting the accesses to the factors or the amount of computation. It was also experimented together with node intervals [4] to RHS with a single nonzero per column, although it was then combined with an interleaving mechanism for parallel issues.

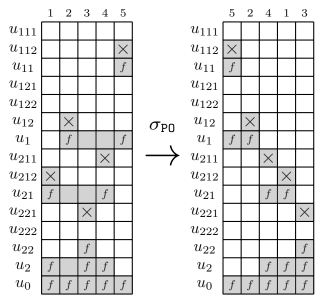
Remark that the RHS B of Figure 5(a) only has one initial nonzero per column. The representative nodes for each column are u_{212}, u_{12}, u_{221}, u_{211}, and u_{112}, respectively. The initial natural order of the columns (INI) induces computation on explicit zeros represented by gray empty cells and we had $\Delta(B) = \Delta(B, \sigma_{\texttt{INI}}) = 948$ and $\Delta_{min}(B) = 744$ (see Example 2.3). On the other hand, the postorder permutation, $\sigma_{\texttt{PO}}$, reorders the columns of B so that the order of their representative nodes u_{112}, u_{12}, u_{211}, u_{212}, u_{221} is compatible with the postordering and avoids computations on explicit zeros. In this case, there are no gray empty cells (see Figure 6(a)) and $\Delta(B, \sigma_{\texttt{PO}}) = \Delta_{min}(B)$. More generally, it can be shown that the postordering induces no extra computations for RHS with a single nonzero per column [4].

For applications with multiple nonzeros per RHS, each column B_j may correspond to a set V_{B_j} with more than one node, among which a representative node should be chosen. We describe two strategies. The first one, called PO_1, chooses as representative node the one corresponding to the first nonzero found in B_j (in the natural order associated to the physical problem). The second one, called PO_2, chooses as representative node in V_{B_j} the one that appears first in the sequence of postordered nodes of the tree. A comparison of the two postorders with the initial natural order is provided in Table 1, for four problems presented in Table 2 of Section 5. Note that the initial order depends on the physical context of the application and has some geometrical properties. Table 1 shows that the choice of the representative node has a significant impact. The superiority of PO_2 over PO_1 is clear and is larger when the number of nonzeros per RHS column is large (problems 5Hz and 7Hz). Indeed, PO_1 is even worse than the initial order on problem 7Hz.

Table 1: Comparison of the number of operations ($\times 10^{13}$) between postorder strategies PO_1 and PO_2.

Δ	INI	PO_1	PO_2	Δ_{min}
H0	.086	.076	**.070**	.050
H3	2.48	1.69	**1.47**	.95
5Hz	.44	.44	**.36**	.22
7Hz	1.46	1.48	**1.21**	.69

Example 3.2. Let $B = [B_1, B_2, B_3, B_4, B_5, B_6] = [\{B_{1,1}, B_{10,1}, B_{19,1}\}, \{B_{4,2}\}, \{B_{13,3}, B_{15,3}\}, \{B_{2,3}\}, \{B_{5,4}, B_{14,4}, B_{22,4}\}, \{B_{10,5}\}]$ be the RHS represented in Figure 6(b). In terms of tree nodes, we have: $V_{B_1} = \{u_{111}, u_{211}, u_0\}$, $V_{B_2} = \{u_{121}\}$, etc. Because the rows of B have already been permuted according to the postordering of the tree, the representative nodes for strategies PO_1 and PO_2 are in both cases the nodes u_{111}, u_{121}, u_{221}, u_{112}, u_{122}, u_{211} (cells with a bold contour), for columns $B_1, B_2, B_3, B_4, B_5, B_6$, respectively. The postorder permutation yields $\sigma_{\texttt{PO}}(B) = [B_1, B_4, B_2, B_5, B_6, B_3]$, which reduces the number of gray cells and the volume of computation with respect to the original column ordering: $\Delta(B) = 1368$ becomes $\Delta(B, \sigma_{\texttt{PO}}) = 1242$. Computations on padded zeros still occur, for example at nodes u_{211} and u_{21} where $\theta(\sigma_{\texttt{PO}}(Z_{u_{211}})) = \theta(\sigma_{\texttt{PO}}(Z_{u_{21}})) = 5$


(a) RHS structure with a single nonzero per column.

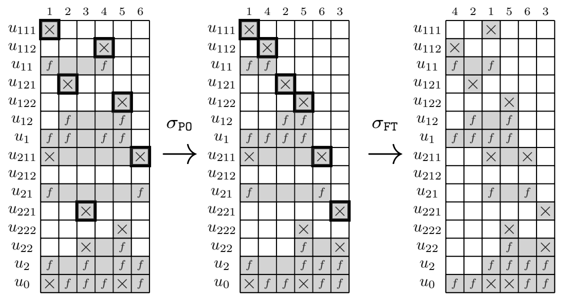

(b) RHS structure with multiple nonzeros per column.

Figure 6: Illustration of the permutation $\sigma_{\texttt{PO}}$ based on a postordering strategy on two RHS with (a) a single initial nonzero per column (Example 2.2), and (b) multiple nonzeros per column (Example 3.2).

whereas $\#Z_{u_{211}} = \#Z_{u_{21}} = 2$. In Figure 6(b), we represented another permutation $\sigma_{\texttt{FT}}$ that will be discussed in the next section and that yields $\Delta(B, \sigma_{\texttt{FT}}) = 1140$.

Remark that the quality of $\sigma_{\texttt{PO}}$ depends on the original postordering. In Example 3.2, if u_{111} and u_{112} were exchanged in the original tree postordering, B_1 and B_4 would be swapped, and Δ would be further improved. A possible drawback of the postorder permutation is also that, since the position of a column is based a single representative node, some information on the RHS structure is unused. We now present a more general and powerful heuristic.

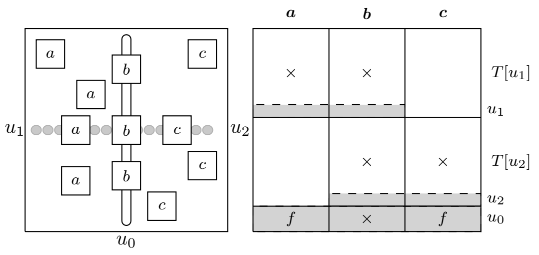
3.2 The Flat Tree permutation

With the aim of satisfying node optimality (see Definition 3.1), we present another algorithm to compute the permutation σ by first illustrating its geometrical properties and then extending it to only rely on algebraic properties.

3.2.1 Geometrical illustration

In the example of Section 1, the variables of a separator u are the ones of the corresponding node u in the tree T. We use the same approach to represent a domain: for $u \in T$, the domain associated with u is defined by the subtree rooted at u and is noted $T[u]$. The set of variables in $T[u]$ corresponds to a subdomain created during the nested dissection algorithm. As an example, the initial 2D domain in Figure 7(a) (left) is $T[u_0]$ and its subdomains created by dividing it with u_0 are $T[u_1]$ and $T[u_2]$. In the following, $T[u]$ will equally refer to a subdomain or a subtree. Figure 7 shows several types of RHS with different positions and nonzero structures. For the sake of simplicity, we assume here that the nonzeros in an RHS column correspond to geometrically contiguous nodes in the domain, as represented in Figure 7(a)(left). We also assume a regular domain for which a perfect nested dissection has been performed. For instance, all separators are in the same direction at each level of the tree.

The *Flat Tree* algorithm relies on the evaluation of the position of each RHS column compared to separators of the nested dissection algorithm. The name Flat Tree comes from the fact that,


(a) Flat Tree step 1.

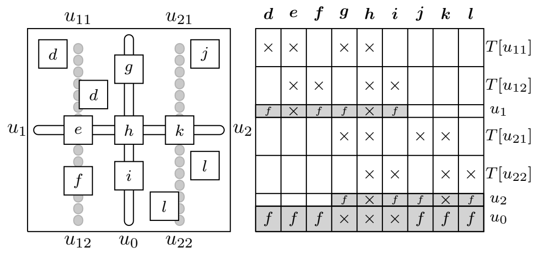

(b) Flat Tree step 2.

Figure 7: A first illustration of the “flat tree” permutation on a 2D domain. In (a) and (b), the figure on the left represents a partitioned 2D domain with different types of RHS, and the one on the right the partial structures of the permuted matrix of RHS. $\times$ or f in a rectangle indicate the presence of nonzeros in the corresponding submatrix, parts of the matrix filled in grey are fully dense and blank parts only contain zeros.

given a parent node with two child subtrees in the separator tree T, the algorithm orders first RHS columns included in the left subtree, then RHS columns associated to the parent (because they intersect both subtrees), and finally, RHS columns included in the right subtree. Figure 7(a) shows the first step of the algorithm: it starts with the root separator u_0 which divides $T = T[u_0]$ into $T[u_1]$ and $T[u_2]$. The initial RHS columns may be *identified* by three different types noted a, b and c according to their positions and nonzero structures. An RHS column is of type a when its nonzero structure is included in $T[u_1]$, c when it is included in $T[u_2]$, and b when it is *divided* by u_0. First, we group the RHS according to their type (a, b, or c) with respect to u_0 which leads to the creation of submatrices/subsets of RHS columns noted $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$. Second, we make sure to place $\boldsymbol{b}$ between $\boldsymbol{a}$ and $\boldsymbol{c}$. We thus achieve operation reduction by guaranteeing node optimality at u_1 and u_2: since all RHS in $\boldsymbol{a}$ and $\boldsymbol{b}$ have at least one nonzero in $T[u_1]$, u_1 belongs to the pruned tree of all of them, hence the dense area filled in gray in the RHS structure. The same is true for $\boldsymbol{b}$ and $\boldsymbol{c}$ and u_2. By permuting B as $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$ ($[\boldsymbol{c}, \boldsymbol{b}, \boldsymbol{a}]$ would also be possible), $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{b}$ and $\boldsymbol{c}$, are contiguous. Thus, $\theta(Z_{u_1}) = \#Z_{u_1}$, $\theta(Z_{u_2}) = \#Z_{u_2}$ and we have u_1- and u_2-optimality. The algorithm proceeds recursively on each newly created submatrix (see Figure 7(b)) to obtain *local* node optimality. First, $\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}$ (resp. $\boldsymbol{j}$, $\boldsymbol{k}$, $\boldsymbol{l}$) form subsets of the RHS of $\boldsymbol{a}$ (resp. $\boldsymbol{c}$) based on their position/type with respect to u_1 (resp. u_2). Second, thanks to the fact that u_1 and u_2 are perfectly aligned, they can be combined to form a single separator that subdivides the RHS of $\boldsymbol{b}$ into three subsets $\boldsymbol{g}$, $\boldsymbol{h}$ and $\boldsymbol{i}$, see Figure 7(b). During this second step, B is permuted as $[\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}]$. The complete permutation is then obtained by applying the algorithm recursively on each subset until the tree is fully processed or the RHS sets contain a single RHS.

This draws the outline of the algorithm introduced with geometrical considerations. The permutation fully results from the position of each RHS with respect to separators. However, the algorithm relies on strong assumptions regarding the ordering algorithm and the RHS structure. Without them, it is difficult or impossible to discriminate RHS columns in many cases (for example, when they are separated by several separators). In order to overcome these limitations and enlarge the application field, we now extend these geometrical considerations with a more general approach.

3.2.2 Algebraic approach

Let us consider the columns of $B = [B_1, B_2, \ldots, B_m]$ as an initially unordered *set* of RHS columns that we note $R_B = \{B_1, B_2, \ldots, B_m\}$. A subset of the columns of B is denoted by $R \subset R_B$ and a generic element of R (one of the columns $B_j \in R$) is noted r. A permuted submatrix of B can be expressed as an ordered *sequence* of RHS columns with square brackets. For two subsets of columns R and R', $[R,\, R']$ denotes a sequence of RHS columns in which the RHS from the subset R are ordered before those from R', without the order of the RHS inside R or R' to be necessarily defined. We found this framework of RHS sets and subsets simpler and better adapted to formalize our algebraic algorithm than matrix notations with complex index permutations. We recall that T is the tree and that, for $r \in R_B$, $T_p(r)$ is the pruned tree of r, as defined in Section 2.1. We now characterize the geometrical position of a RHS using the notion of *pruned layer*: for a given depth d in the tree, and for a given RHS r, we define the *pruned layer* $L_d(r)$ as the set of nodes at depth d in the pruned tree $T_p(r)$. In the example of Figure 7(a), $L_1(r) = \{u_1\}$ for all $r \in \boldsymbol{a}$, $L_1(r) = \{u_2\}$ for all $r \in \boldsymbol{c}$, and $L_1(r) = \{u_1, u_2\}$ for all $r \in \boldsymbol{b}$. The notion of pruned layers allows to formally identify sets of RHS with common characteristics in the tree without any geometrical information. This is formalized and generalized by Definition 3.3.

Definition 3.3. Let $R \subset R_B$ be a set of RHS, and let U be a set of nodes at depth d of the tree T. We defined $R[U] = \{r \in R \mid L_d(r) = U\}$ as the subset of RHS with pruned layer U.

We have for example, see Figure 7: $R[\{u_1\}] = \boldsymbol{a}, R[\{u_2\}] = \boldsymbol{c}$ and $R[\{u_1, u_2\}] = \boldsymbol{b}$ at depth $d = 1$.

The algebraic recursive algorithm is depicted in Algorithm 1. Its arguments are R, a set of RHS and d, the current depth. Initially, $d = 0$ and $R = R_B = R[u_0]$, where u_0 is the root of the tree T. At each step of the recursion, the algorithm builds the distinct pruned layers $U_i = L_{d+1}(r)$ for the RHS r in R. Then, instead of looking for a permutation σ to minimize $\sum_{u \in T_p(R_B)} \delta_u \times \theta(\sigma(Z_u))$ (10), it orders the $R[U_i]$ by considering the *restriction* of problem (10) to R and to nodes at depth $d+1$ of $T_p(R)$. Furthermore, with the assumption that T is balanced, all nodes at a given level of $T_p(R)$ are of comparable size. δ_u may thus be assumed *constant* per level and needs not be taken into account in our minimization problem. The algorithm is thus a greedy top-down algorithm, where at each step a local optimization problem is solved. This way, priority is given to the top layers of the tree, which are in general more critical because factor matrices are larger.

Algorithm 1 Flat Tree

```
procedure FLATTREE($R$, $d$)
    1) Build the set of children $C(R)$
    1.1) Identify the distinct pruned layers (pruned layer = set of nodes)
    $\mathcal{U} \leftarrow \emptyset$
    for all $r \in R$ do
        $\mathcal{U} \leftarrow \mathcal{U} \cup \{L_{d+1}(r)\}$
    end for
    1.2) $C(R) = \{R[U] \mid U \in \mathcal{U}\}$
    2) Order children $C(R)$ as $[R[U_1], \ldots, R[U_{\#C(R)}]]$:
    return [FLATTREE($R[U_1], d+1$),...,FLATTREE($R[U_{\#C(R)}], d+1$)]
end procedure
```

The recursive structure of the algorithm can be represented by a recursion tree T_{rec} defined as follows: each node R of T_{rec} represents a set of RHS, $C(R)$ denotes the set of children of R and the root is R_B. By construction of Algorithm 1, $C(R)$ is a partition of R, *i.e.*, $R = \dot{\bigcup}_{R' \in C(R)} R'$ (disjoint union). Note that all $r \in R$ such that $L_{d+1}(r) = \emptyset$ belong to $R[\emptyset]$, which is also included in $C(R)$. In this special case, $R[\emptyset]$ can be added at either extremity of the current sequence without introducing extra computation and the recursion stops for those RHS, as will be illustrated in Example 3.3.

With this construction, each leaf of T_{rec} contains RHS with indistinguishable nonzero structures, and keeping them contiguous in the final permutation avoids introducing extra computations. Assuming that for each $R \in T_{rec}$ the children $C(R)$ are ordered, this induces an ordering of all the leaves of the tree, which defines the final RHS sequence. We now explain how the set of children $C(R)$ is built and ordered at each step:

1) Building the set of children The set of children of R in the recursion tree is built by first identifying the pruned layers U of all RHS $r \in R$. The different pruned layers are stored in $\mathcal{U}$ and we have for example (Figure 7, first step of the algorithm), $\mathcal{U} = \{\{u_1\}, \{u_2\}, \{u_1, u_2\}\}$. Using Definition 3.3, we then define $C(R) = \{R[U] \mid U \in \mathcal{U}\}$, which forms a partition of R. One important property is that all $r \in R[U]$ have the same nonzero structure at the corresponding layer so that numbering them contiguously prevent the introduction of extra computation.

2) Ordering the children At each depth d of the recursion, the ordering of the children results from the resolution of the local optimization problem consisting in finding a sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$ such that the size of the intervals is minimized for all nodes u at depth $d+1$ of the pruned tree $T_p(R)$. As mentioned earlier, when solving this local optimization problem, the RHS order inside each $R[U_i]$ has no impact on the size of the intervals (it will only impact lower levels). For any node u in $T_p(R)$ such that $depth(u) = d+1$, the size of the interval is then:

$$\theta(Z_u|_R) = \max(Z_u|_R) - \min(Z_u|_R) + 1 = \sum_{i=i_{\min}(u)}^{i_{\max}(u)} \#R[U_i],$$

where $Z_u|_R$ is the set of permuted indices representing the active columns restricted to R, and $i_{\min}(u) = \min\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$ (resp. $i_{\max}(u) = \max\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$) is the first (resp. last) index i such that $u \in U_i$.

Proof. In the sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$, $\min(Z_u|R)$ (resp. $\max(Z_u|_R)$) corresponds to the index of the first (resp. last) column in $R[U_{i_{min}}]$ (resp. $R[U_{i_{max}}]$). Since all columns from $R[U_{i_{min}}]$ to $R[U_{i_{max}}]$ are numbered consecutively, we have the desired result. □

Finally, our local problem consists in minimizing the local cost function (sum of the interval sizes for each node at depth $d+1$):

$$\text{cost}([R[U_1], \ldots, R[U_{\#C(R)}]]) = \sum_{\substack{u \in T_p(R) \\ depth(u)=d+1}} \sum_{i=i_{min}(u)}^{i_{max}(u)} \#R[U_i] \tag{11}$$

To build the ordered sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$, we use a greedy algorithm that starts with an empty sequence, then, at each step $k \in \{1, \ldots, \#C(R)\}$, we insert a RHS set $R[U]$ picked randomly in $C(R)$ at the position that minimizes (11) on the current sequence. To do so, we simply start from one extremity of the sequence of size $k-1$ and compute (11) for the new sequence of size k for each possible position $0 \dots k$; if several positions lead to the same minimal cost, the first one encountered is chosen. In case $u-$optimality is obtained for each node u considered, then the permutation is said to be *perfect* and the cost function is minimal, locally inducing no extra operations on those nodes.

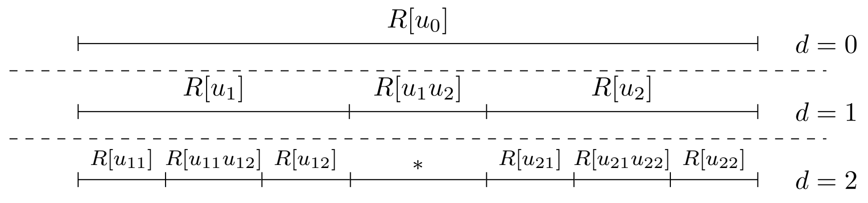
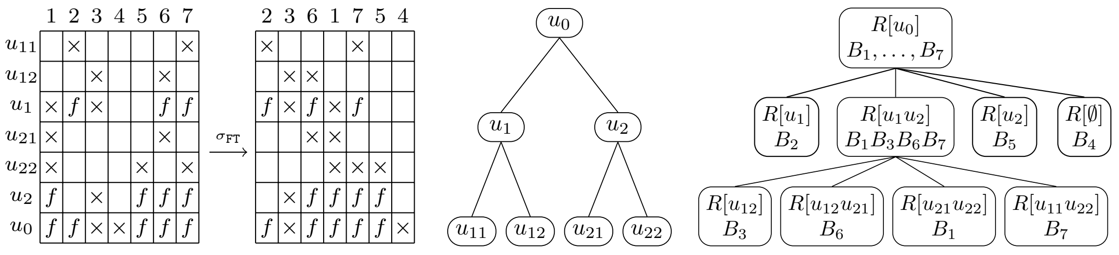


Figure 8: Representation of a layered sequence built by the Flat Tree algorithm on a binary tree. Sets with empty pruned layers have not been represented but could be added at the extremity of the concerned sequence (e.g., right after $R[u_2]$ for a RHS included in u_0). With the geometric assumptions corresponding to Figure 7, one would have * = $R[u_{11}u_{21}]$, $R[u_{11}u_{12}u_{21}u_{22}]$, $R[u_{12}u_{22}]$. Without such assumptions, the sequence * is more complex.

Figure 8 shows the recursive structure of the RHS sequence after applying the algorithm on a binary tree. We refer to this representation as the *layered sequence*. For simplicity, the notation for pruned layers has been reduced from, e.g., $\{u_1\}$ to u_1, and from $\{u_1, u_2\}$ to u_1u_2. From the recursion tree point of view, $R[u_1], R[u_1u_2], R[u_2]$ are the children of $R[u_0]$ in T_{rec}, $R[u_{11}], R[u_{11}u_{12}], R[u_{12}]$ the ones of $R[u_1]$, etc.



(a) RHS structure. (b) Separator tree T. (c) Recursion tree T_{rec}.

Figure 9: Illustration of the algebraic Flat Tree algorithm on a set of 7 RHS.

Example 3.3. Let $B = [B_1, B_2, B_3, B_4, B_5, B_6, B_7]$ be a RHS matrix with the structure presented in Figure 9(a). Although we still use a binary tree, we make no assumption on the RHS structure, on the domain, or on the ordering. We have $R_B = R[u_0] = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$. At depth 1, the set of pruned layers corresponding to $R[u_0]$ is $\mathcal{U} = \{u_1, u_1u_2, u_2, \emptyset\}$, with the RHS partition composed of the sets $R[u_1], R[u_2], R[u_1u_2]$ and $R[\emptyset]$. Then, $C(R[u_0]) = \{R[u_1], R[u_1u_2], R[u_2], R[\emptyset]\}$ and the recursion tree shown in Figure 9(c) is built from top to bottom. As can be seen in the non-permuted RHS structure, $R[\emptyset] = B_4$ at depth 1 induces extra operations at nodes descendant of u_0, which disappear when placing $R[\emptyset]$ at one extremity of the sequence. We choose to place it last, and the ordered sequence obtained by the algorithm is $[R[u_1], R[u_1u_2], R[u_2], R[\emptyset]]$ ($[R[u_2], R[u_1u_2], R[u_1], R[\emptyset]]$ is also possible). A recursive call is done on each of the identified sets. We only focus on $R = R[u_1u_2]$ since $R[u_1]$, $R[u_2]$ and $R[\emptyset]$ contain a single RHS which needs not be further ordered. At this stage, the set of pruned layers is $\mathcal{U} = \{u_{21}u_{22}, u_{12}, u_{12}u_{21}, u_{11}u_{22}\}$. It appears that the sequence $[R[U_1], R[U_2], R[U_3], R[U_4]]$, where $U_1 = u_{12}$, $U_2 = u_{12}u_{21}$, $U_3 = u_{21}u_{22}$, and $U_4 = u_{11}u_{22}$ is a perfect sequence which gives local optimality. However, taking the problem globally, we see that $\theta(Z_{u_{11}}) \neq \#Z_{u_{11}}$ with the final sequence $[B_2, B_3, B_6, B_1, B_7, B_5, B_4]$.

The algebraic algorithm simplifies the assumptions that were made in the geometrical one. The nonzeros of each RHS no longer need to be geometrically localized, and we can address irregular problems and orderings that yield non binary trees. We compared both approaches on problems H0, H3, 5Hz and 7Hz from Table 2 and observed, even with nested dissection, an average 7% gain on Δ with the algebraic approach, which we will use in all our experiments. Nevertheless, computations on explicit zeros (for example zero rows in column f and subdomain $T[u_{11}]$ in Figure 7(b)), may still occur. This will also be illustrated in Section 5, where $\Delta(B, \sigma_{\texttt{FT}})$ is 39% larger than $\Delta_{min}(B)$, in the worst case. A Blocking algorithm is now introduced to further reduce $\Delta(B, \sigma_{\texttt{FT}})$.

4 Toward a minimal number of operations using blocks

In this section, we identify the causes of the remaining extra operations and provide an efficient blocking algorithm to reduce them efficiently while creating a small number of blocks. The algorithm relies on a property of independence of right-hand sides that is first illustrated, and then formalized.

4.1 Objectives and first illustration of independence property

The use of blocking techniques may fulfill different objectives. In terms of operation count, optimality ($\Delta_{min}(B)$) is obtained when processing the columns of B one by one, which implies the creation of m blocks. However, this requires processing the tree m times and will typically lead to a poor arithmetic intensity (and likely a poor performance). On the other hand, the algorithms of Section 3 only use one block, which allows a higher arithmetic intensity but leads to extra operations. In the dense case, blocks are also often used to improve the arithmetic intensity. In the sparse RHS case, blocking techniques with regular blocks of columns have been associated to tree pruning to either limit the access to the factors [3], or limit the number of operations [18]. They were either based on a preordering of the columns or on hypergraph models. In this section, to give as much flexibility as possible to the underlying algorithms and avoid unnecessary constraints, our objective is to create a minimal number of (possibly large) blocks while reducing the number of extra operations by a given amount. In particular, we allow blocks to be irregular and assume node intervals are exploited within each block.

On the one hand, in the same way as variables in two different domains are independent, two RHS or two sets of RHS included in two different domains exhibit interesting properties, as can be observed for sets $\boldsymbol{a} \in T[u_1]$ and $\boldsymbol{c} \in T[u_2]$ from Figure 7(a). It implies that no extra operations are introduced between them: $\Delta([\boldsymbol{a}, \boldsymbol{c}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{a}$ and $\boldsymbol{c}$ are *independent sets* and can thus be associated together. On the other hand, a set of RHS intersecting a separator (such as set $\boldsymbol{b}$) exhibits some zeros and nonzeros in rows common to their adjacent RHS sets ($\boldsymbol{a}$ and $\boldsymbol{c}$) which will likely introduce extra computation. For example, one can see in Figure 7(b) that $\Delta([\boldsymbol{a}, \boldsymbol{b}]) = \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) > \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}]) + \Delta([\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b})$ and that $\Delta([\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]) >$

$\Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{b}$ is a set of *problematic RHS*. Another example is the one of Figure 6 (right), where extracting the problematic RHS B_1 and B_5 from $[B_4, B_2, B_1, B_5, B_6, B_3]$ suppresses all extra operations: $\Delta([B_4, B_2, B_6, B_3]) + \Delta([B_1, B_5]) = \Delta_{min} = 1056$.

To give further intuition on the Blocking algorithm, consider the RHS structure of Figure 7(b). Problematic RHS $\boldsymbol{e}$ and $\boldsymbol{k}$ in $[\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}]$ can be extracted to form two blocks, or *groups*, $[\boldsymbol{e}, \boldsymbol{k}]$ and $[\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]$. The situation is slightly more complicated for $[\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]$, where $\boldsymbol{h}$ indeed intersects two separators, u_1 and u_2. In this case, $\boldsymbol{h}$ should be extracted to form the groups $[\boldsymbol{g}, \boldsymbol{i}]$ and $[\boldsymbol{h}]$. We note that the amount of extra operations will likely be much larger when the separator intersected is high in the tree. Situations where no assumption on the RHS structure is made are more complicated and require a more general approach. For this, we formalize the notion of *independence*, which will be the basis for our blocking algorithm.

4.2 Algebraic formalization and first blocking algorithm

In this section, we give a first version of the Blocking algorithm. It is based on a sufficient condition allowing to group together sets of RHS without introducing extra computation. We assume the matrix B to be flat tree ordered and the recursion tree T_{rec} to be built and ordered. Using the notations of Definition 3.3, we first give an algebraic definition of the independence property between two sets of RHS:

Definition 4.1. Let U_1, U_2 be two sets of nodes at a given depth of a tree T, and let $R[U_1], R[U_2]$ be the corresponding sets of RHS. $R[U_1], R[U_2]$ are said to be *independent* if and only if $U_1 \cap U_2 = \emptyset$.

With Definition 4.1, we are able to formally identify independent sets and we will show formally why they can be associated together. For example, take $\boldsymbol{a} = R[u_1]$ and $\boldsymbol{c} = R[u_2]$ from Figure 7(a), $R[u_1]$ and $R[u_2]$ are independent and $\Delta([R[u_1], R[u_2]]) = \Delta(R[u_1]) + \Delta(R[u_2])$. On the contrary, when $R[U_1], \ldots, R[U_n]$ are not pairwise independent, the objective is to group together independent sets of RHS, while forming as few groups as possible. In terms of graphs, this problem is equivalent to a classical coloring problem, where $R[U_1], \ldots, R[U_n]$ are the vertices and an edge exists between $R[U_i]$ and $R[U_j]$ if and only if $U_i \cap U_j \neq \emptyset$. Several heuristics exist for this problem, and each color will correspond to one group. The Blocking algorithm as depicted in Figure 10 consists of a

Algorithm 2 Blocking algorithm

```
for $d = 0$ to $d_{max}$ do
    $j \leftarrow 0$ /* #groups at depth $d+1$ */
    for all groups $g_i^d$ at detph $d$ do
        $(g_{j+1}^{d+1} \ldots g_{j+k}^{d+1}) \leftarrow$ BuildGroups($g_i^d, d+1$)
        /* $k$ new groups have been created */
        $j \leftarrow j + k$
    end for
end for
```

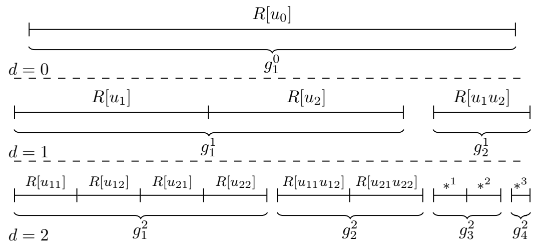


Figure 10: A first version of the Blocking algorithm (left). It is illustrated (right) on the layered sequence of Figure 8. With the geometric assumptions of Figure 7, $*^1 = R[u_{11}u_{21}], *^2 = R[u_{12}u_{22}]$, and $*^3 = R[u_{11}u_{12}u_{21}u_{22}]$.

top-down traversal of T_{rec} where at each depth d and for each intermediate group g_i^d, the central procedure BuildGroups is called. Notice that any group g_i^d verifies the following properties: (i) g_i^d can be represented by a sequence $[R[U_1], \dots, R[U_n]]$, and (ii) the sequence respects the flat tree order of T_{rec}. Then, BuildGroups(g_i^d, $d+1$) first builds the sets of RHS at depth $d+1$, which are exactly the children of the $R[U_j] \in g_i^d$ in T_{rec}. Second, BuildGroups(g_i^d, $d+1$) solves the aforementioned coloring problem on these RHS sets and builds the k groups $(g_{j+1}^{d+1}, \dots, g_{j+k}^{d+1})$.

In the example of Figure 10(right), there is initially a single group $g_1^0 = [R[u_0]]$ with one set of RHS. This group may be expressed as the ordered sequence $[R[u_1]R[u_1u_2]R[u_2]]$, since $C(R[u_0]) = \{R[u_1], R[u_1u_2], R[u_2]\}$. g_1^0 does not satisfy the independence property at depth 1 because $u_1 \cap u_1u_2 \neq \emptyset$ or $u_2 \cap u_1u_2 \neq \emptyset$. BuildGroups($g_1^0$, 1) yields $g_1^1 = [R[u_1], R[u_2]]$ and $g_2^1 = [R[u_1u_2]]$. The algorithm proceeds on each group until a maximal depth d_{max} is reached: $(g_1^2, g_2^2) =$ BuildGroups$(g_1^1, 2)$, $(g_3^2, g_4^2) =$ BuildGroups$(g_2^1, 2)$, etc. To illustrate the interest of property (ii) on groups, let us take sets $\boldsymbol{d} = R[u_{11}], \boldsymbol{f} = R[u_{12}], \boldsymbol{j} = R[u_{21}]$ and $\boldsymbol{l} = R[u_{22}]$ from Figure 7(b). One can see that $\Delta([\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]) = \Delta(\boldsymbol{d}) + \Delta(\boldsymbol{f}) + \Delta(\boldsymbol{j}) + \Delta(\boldsymbol{l}) < \Delta([\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}])$. Compared to $[\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]$ which respects the global flat tree ordering and ensures u_1- and u_2-optimality, $[\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}]$ does not and thus increases $\theta(Z_{u_1})$ and $\theta(Z_{u_2})$.

Furthermore, Algorithm 2 ensures the following property, which shows that the independent sets of RHS grouped together do not introduce extra operations.

Property 4.1. *For any group $g^d = [R[U_1], \dots, R[U_n]]$ created through Algorithm 2 at depth d, we have $\Delta([R[U_1], \dots, R[U_n]]) = \sum_{i=1}^n \Delta(R[U_i])$.*

Proof. For $d \geq 1$, let $g^d = [R[U_i^d]_{i=1,\dots,n^d}]$ be a group at depth d created through Algorithm 2 (we use superscripts d in this proof to indicate the depth without ambiguity). Let us split nodes above (A) and below (B) layer d in the pruned tree $T_p(g^d)$. The number of operations to process g^d is:

$$\Delta(g^d) = \sum_{u \in T_p(g^d)} \delta_u \times \theta(Z_u|_{g^d}) = \overbrace{\sum_{u \in A} \delta_u \times \theta(Z_u|_{g^d})}^{\Delta_A} + \overbrace{\sum_{u \in B} \delta_u \times \theta(Z_u|_{g^d})}^{\Delta_B}, \tag{12}$$

where $A = \{u \in T_p(g^d) \mid depth(u) < d\}$ and $B = \{u \in T_p(g^d) \mid depth(u) \geq d\}$.
(i) We first consider the term Δ_B. Let $B_i = \{u \in T_p(R[U_i^d]) \mid depth(u) \geq d\}$. Thanks to the independence property of the $R[U_i^d]$ forming g^d, the pruned layers U_i^d in T are disjoint and since T is a tree, we have $B_i \cap B_j = \emptyset$ for all $i \neq j$. Hence, $B = \dot{\bigcup}_{i=1}^{n^d} B_i$, where $\dot{\bigcup}$ denotes the disjoint union. Therefore,

$$\Delta_B = \sum_{u \in \dot{\bigcup}_{i=1}^{n^d} B_i} \delta_u \times \theta(Z_u|_{[R[U_j^d]_{j=1,\dots n^d}]}) = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{[R[U_j^d]_{j=1,\dots n^d}]}).$$

We recall that a RHS r is said to be active at node u if $u \in T_p(r)$. In the inner sum, the only possible active RHS in B_i are the ones that belong to $R[U_i^d]$ (independence of the $R[U_j^d]$), so that for all $u \in B_i$, we have $\theta(Z_u|_{[R[U_i^d]_{i=1,\dots n^d}]}) = \theta(Z_u|_{R[U_i^d]})$. Therefore, $\Delta_B = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{R[U_i^d]})$.
(ii) We now consider the term Δ_A. Similarly to (i), we define $A_i = \{u \in T_p(R[U_i^d]) \mid depth(u) < d\}$. We have $A = \bigcup_{i=1}^{n_d} A_i$ but the union is no longer disjoint. Let $T_{rec}(g^d)$ be the restriction to g^d of the recursion tree T_{rec} associated to the flat-tree algorithm applied to R_B (see Section 3.2.2 for the definition of T_{rec}). $T_{rec}(g^d)$ is obtained by excluding at each node of T_{rec} the right-hand sides that

are not part of g^d, then by pruning all empty nodes. We also restrict Definition 3.3 to g^d and thus note $R[U] = \{r \in g^d \mid L_d(r) = U\}$. In particular, the root of $T_{rec}(g^d)$ is $R[u_0] = g^d$.

By construction of Algorithm 2 (Figure 10), we know that any layer at depth $d' < d$ of the group g^d consists of independent sets $R[U_j^{d'}]$ of RHS. Therefore, $\forall u \in A$, $\exists! R[U] \in T_{rec}(g^d)$ such that $u \in U$. This means that the only active columns at node u are those in this unique $R[U]$ and, since the RHS in $R[U]$ are all contiguous in g^d thanks to the global flat tree ordering, we have $\theta(Z_u|_{R[U]}) = \theta(Z_u|_{g^d}) = \#R[U]$.

Furthermore, by construction of the recursion tree (children nodes form a partition of each parent node), the RHS in $R[U]$ are the ones in the disjoint union of $R[U_i^d] \subset R[U]$, the sets of right-hand sides at layer d that are descendants of $R[U]$ in $T_{rec}(g^d)$. Therefore, $\#R[U] = \sum_{R[U_i^d] \subset R[U]} \#R[U_i^d]$. Furthermore, since the $R[U_i^d]$ such that $R[U_i^d] \subset R[U]$ are contiguous sets in g^d and are all active at node u, we also have $\theta(Z_u|_{R[U_i^d]}) = \#R[U_i^d]$. It follows:

$$\theta(Z_u|_{g^d}) = \sum_{R[U_i^d] \subset R[U]} \theta(Z_u|_{R[U_i^d]}).$$

We define $\xi_i(u) = 1$ if $R[U_i^d] \subset R[U]$ (with $R[U]$ derived from u as explained above), and $\xi_i(u) = 0$ otherwise. The condition $R[U_i^d] \subset R[U]$ means that u is an ancestor of U_i^d nodes in T. Thus, $\xi_i(u) = 1$ for $u \in A_i$ and $\xi_i(u) = 0$ for $u \notin A_i$. We can thus write $\sum_{R[U_i^d] \subset R[U]} \theta(Z_u|_{R[U_i^d]}) = \sum_{i=1}^{n^d} \xi_i(u) \theta(Z_u|_{R[U_i]})$ and redefine Δ_A as:

$$\begin{aligned}
\Delta_A = \sum_{u \in A} \delta_u \times \theta(Z_u|_{g^d}) &= \sum_{u \in A} \delta_u \times \sum_{i=1}^{n^d} \xi_i(u) \theta(Z_u|_{R[U_i^d]}) \\
&= \sum_{i=1}^{n^d} \sum_{u \in A} \delta_u \times \xi_i(u) \theta(Z_u|_{R[U_i^d]}) \\
&= \sum_{i=1}^{n^d} \sum_{u \in A_i} \delta_u \times \theta(Z_u|_{R[U_i^d]}).
\end{aligned}$$

Joining the terms Δ_A and Δ_B, we finally have:

$$\Delta(g^d) = \Delta_A + \Delta_B = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{R[U_i]}) + \sum_{i=1}^{n^d} \sum_{u \in A_i} \delta_u \times \theta(Z_u|_{R[U_i]}) = \sum_{i=1}^{n^d} \Delta(R[U_i])$$

□

Interestingly, Property 4.1 can be used to prove, in the case of a single nonzero per RHS, the optimality of the Flat Tree permutation.

Corollary 4.1. *Let R_B be the initial set of RHS such that $\forall r \in R_B, \#V_r = 1$. Then the Flat Tree permutation is optimal: $\Delta(R_B) = \Delta_{min}(R_B)$.*

Proof. Since $\forall r \in R_B, \#V_r = 1$, $T_p(r)$ is a branch of T. Indeed, V_r is a singleton and $T_p(r)$ is built by following the path in T from V_r up to the root. As a consequence, any set of RHS $R[U]$ built through the Flat Tree algorithm will be represented by a pruned layer U containing a single node u. Thus, at each step of the algorithm, the RHS sets identified by the Flat Tree algorithm are all independent from each other. In case Algorithm 2 is applied, a unique group R_B is then kept until the bottom of the tree. Blocking is thus not needed and Property 4.1 applies at each level of the flat tree recursion. $\Delta(R_B)$ is thus equal to the sum of the $\Delta(R[U])$ for all leaves $R[U]$ of the recursion tree T_{rec}. Since $\Delta(R[U]) = \Delta_{min}(R[U])$ on those leaves (all RHS in $R[U]$ involve the exact same nodes and operations), we conclude that $\Delta(R_B) = \Delta_{min}(R_B)$. □

This proof is independent of the specific ordering of the children at step 2 of Algorithm 1. The corollary is therefore more general: any recursive top-down ordering based on keeping together at each layer the RHS with an identical pruned layer is optimal, as long as the pruned layers identified at each layer are independent.

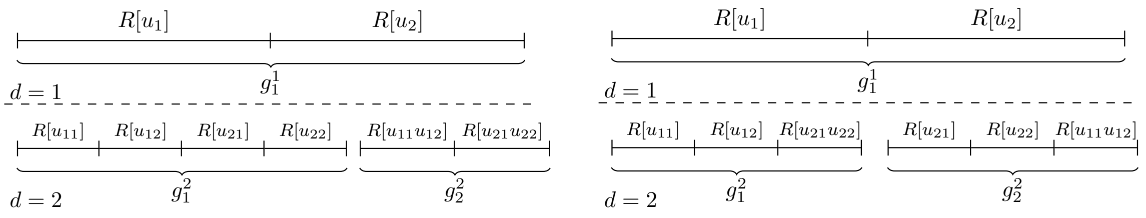


Figure 11: Two strategies to build groups: CritPathBuildGroups (left) and RegBuildGroups (right).

Back to the BuildGroups function and the coloring problem, we mention that the solution may not be unique. Even on the simple example of Figure 10, there are several ways to define groups, as shown in Figure 11 for g_1^1: both strategies satisfy the independence property and minimize the number of groups. The CritPathBuildGroups strategy tends to create a large group g_1^2 and a smaller one, g_2^2. In each group the computations on the tree nodes are expected to be well balanced because all branches of the tree rooted at u_0 might be covered by the RHS (assuming thus a reasonably balanced RHS distribution over the tree). The choice of CritPathBuildGroups can be driven by tree parallelism considerations, namely, the limitation of the sum of the operation count on the *critical paths* of all groups. The RegBuildGroups strategy tends to balance the sizes of the groups but may create more unbalance regarding the distribution of work over the tree.

We note that for a given depth, the application of BuildGroups on *all* groups may lead to a too rapid and unnecessary increase of the number of groups. Furthermore, the enforcement of the independence property during the BuildGroups operation may require the creation of more than two groups. In the next section we add features to minimize the number of groups created using greedy heuristics and propose our final version of the Blocking algorithm.

4.3 A greedy approach to minimize the number of groups

The final greedy blocking algorithm is given by Algorithm 3. Compared to Algorithm 2, it adds the group selection, limits the number of groups during BuildGroups to two, and stops when a given tolerance on the amount of extra operations is reached.

First, instead of stepping into each group, as in Algorithm 2, the group selection consists in choosing among the current groups the one responsible for most extra computation, that is, the one for which $\Delta(g) - \Delta_{min}(g)$ is maximal. This implies that groups that are candidate for splitting might have been created at different depths and we use a superscript to indicate the depth d at which a group was split, as in the notation g_0^d.

Second, enforcing the independence property inside all the groups created may be inappropriate for some problems, because the number of groups may increase much more than needed. Instead of a coloring problem, BuildGroups (called BuildMaxIndepSet) now looks inside the sets of g_0^d for a maximal group of independent sets of RHS at depth $d+1$, noted g_{imax}^{d+1}, and leaves the other sets in another group g_c^d, whose depth remains equal to d. The latter may thus consist of dependent sets that may be subdivided later if needed[2]. Rather than an exact algorithm to determine g_{imax}, we use a greedy heuristic that forms a *maximal* independent set.

Finally, we define μ_0 as the tolerance of extra operations authorized. With a typical value $\mu_0 = 1.01$, the algorithm stops when the number of extra operations is within 1% of the minimal number of operations, Δ_{min}. When the algorithm stops, G contains the final set of groups.

Algorithm 3 Blocking algorithm

```
$G \leftarrow \{R_B\}$, $\Delta_{min} \leftarrow \Delta_{min}(R_B)$, $\Delta \leftarrow \Delta(R_B)$
while $\Delta/\Delta_{min} > \mu_0$ do
    Select $g_0^d$ such that $\Delta(g_0^d) - \Delta_{min}(g_0^d) = \max_{g \in G} (\Delta(g) - \Delta_{min}(g))$    ▷ Group selection
    $(g_{imax}^{d+1}, g_c^d) \leftarrow$ BuildMaxIndepSet($g_0^d$, d+1)
    $G \leftarrow G \cup \{g_{imax}^{d+1}, g_c^d\} \setminus \{g_0^d\}$
    $\Delta \leftarrow \Delta - \Delta(g_0^d) + \Delta(g_{imax}^{d+1}) + \Delta(g_c^d)$
end while
```

5 Experimental results

In this section, we report on the operation count Δ resulting from the proposed permutation and blocking algorithms, measured in terms of number of operations during the forward elimination (Equation (2)). Our experiments are performed on a set of 3D regular finite difference problems coming from seismic and electromagnetism modeling [1, 16], for which the cost of the solve phase is critical. The characteristics of the corresponding matrices and associated RHS are presented in Table 2. In both applications, the nonzeros of each RHS correspond to a small set of close points, near the top of the 3D grid corresponding to the physical domain, and there is some overlap between RHS. Except in Section 5.3, a geometric nested dissection (ND) algorithm is used to reorder the matrix.

5.1 Impact of the Flat Tree algorithm

We first introduce the terminology used to denote the different strategies developed in this study and that impact the number of operations Δ. `DEN` represents the dense case, where no optimization is used to reduce Δ, and `TP` means tree pruning. When column intervals are exploited at each tree node, we denote by `RAN`, `INI`, `PO` and `FT` the random, initial ($\sigma = id$), Postorder ($\sigma_{\texttt{PO}}$) and Flat Tree ($\sigma_{\texttt{FT}}$) permutations, respectively.

[2]In case g_c^d consists of independent sets and is selected, the exact same sets will be used for g_c^{d+1}, which will then only be subdivided at depth $d+2$.

Table 2: Characteristics of the $n \times n$ matrix A and $n \times m$ matrix B for different test cases. $D(A) = nnz(A)/n$ and $D(B) = nnz(B)/m$ represent the average number of nonzeros per column of A and B, respectively.

application	matrix	$n(\times 10^6)$	$D(A)$	sym	m	$D(B)$
seismic modeling	5Hz	2.9	24	no	2302	567
	7Hz	7.2	25	no	2302	486
	10Hz	17.2	26	no	2302	486
electro-magnetism modeling	H0	.3	13	yes	8000	9.8
	H3	2.9	13	yes	8000	7.5
	H17	17.4	13	yes	8000	6
	H116	116.2	13	yes	8000	6
	S3	3.3	13	yes	12340	19.7
	S21	20.6	13	yes	12340	9.5
	S84	84.1	13	yes	12340	8.6
	D30	29.7	23	yes	3914	7.6

Table 3: Number of operations ($\times 10^{13}$) during the forward elimination ($LY = B$) according to the strategy used (ND ordering).

Δ	DEN	TP	RAN	INI	PO	FT	Δ_{min}
5Hz	1.73	.74	.74	.44	.36	**.28**	.22
7Hz	5.94	2.54	2.52	1.46	1.21	**.92**	.69
10Hz	20.62	9.01	8.92	4.78	3.85	**2.87**	2.26
H0	0.39	0.11	0.11	0.086	0.070	**0.057**	0.050
H3	7.19	3.33	3.31	2.48	1.47	**1.26**	0.95
H17	81.34	37.15	36.97	27.52	10.41	**10.21**	10.12
H116	990.02	448.31	445.91	327.89	123.79	**121.76**	120.68
S3	13.36	4.98	4.91	3.73	2.65	**2.17**	1.71
S21	156.20	49.04	48.07	35.42	25.73	**22.53**	19.43
S84	983.48	286.57	282.70	222.59	161.87	**138.56**	118.51
D30	71.60	39.78	39.38	19.49	10.93	**10.21**	7.31

Table 4: Theoretical tree parallelism according to the strategy used (ND ordering).

S	DEN	TP	RAN	INI	PO	FT
5Hz	8.60	3.91	3.88	3.11	2.39	**2.54**
7Hz	8.92	3.97	3.94	3.02	2.25	**2.48**
10Hz	9.10	4.04	4.02	2.96	2.30	2.30
H0	5.88	2.11	2.11	1.75	1.51	1.45
H3	5.99	3.22	3.21	2.47	2.02	2.11
H17	6.32	3.34	3.32	2.54	2.00	1.97
H116	7.92	3.63	3.61	2.75	2.05	2.02
S3	6.12	2.84	2.83	2.18	1.73	1.61
S21	6.30	2.56	2.46	1.85	1.49	1.47
S84	8.01	2.41	2.38	1.90	1.53	1.52
D30	8.50	4.73	4.70	2.86	2.05	2.56

Table 5: Impact of the number of groups NG on the normalized operation count, until Δ_{NG}/Δ_{min} becomes smaller than the tolerance $\mu_0 = 1.01$ (ND ordering).

Δ_{NG}/Δ_{min}	FT	NG= 2	NG= 3	NG= 4	NG= 5
5Hz	1.283	1.111	1.001	x	x
7Hz	1.321	1.116	1.002	x	x
10Hz	1.269	1.029	1.002	x	x
H0	1.148	1.029	1.010	1.002	x
H3	1.329	1.068	1.027	1.005	x
H17	1.009	x	x	x	x
H116	1.009	x	x	x	x
S3	1.275	1.120	1.045	1.012	1.003
S21	1.160	1.037	1.015	1.003	x
S84	1.169	1.041	1.015	1.002	x
D30	1.397	1.082	1.058	1.024	1.004

The improvements brought by the different strategies are presented in Table 3. Compared to the dense case, TP divides Δ by at least a factor 2. In the case column intervals are exploited at each node, the large gap between RAN and INI shows that the original column order holds geometrical properties. FT behaves better than INI and PO and gets reasonably close to Δ_{min}. Overall, FT provides a 13% gain on average over PO. However, the gain on Δ decreases from 25% on the 10Hz problem to 1% on the H116 problem. This can be explained by the fact that B is denser for the seismic applications than for the electromagnetism applications (see Table 2). Indeed, the sparser B, the closer we are from a single nonzero per RHS in which case both FT and PO are optimal.

Second, we evaluate the impact of exploiting RHS sparsity on tree parallelism. For this, we report in Table 4 the maximal theoretical speed-up S that can be reached using tree parallelism only (node parallelism is also needed, for example on the root). It is defined as $S = \frac{\Delta}{\Delta_{cp}}$ where Δ_{cp} is the number of operations on the critical path of the tree. We observe that tree parallelism is significantly smaller than in the dense case. This is because the depth of the pruned tree $T_p(B)$ is similar to the one of the original tree (some nonzeros of B appear in general in the leaves), while the tree effectively processed is pruned and thus the overall amount of operations is reduced. For the same reason, S is smaller for test cases where $nnz(B)/m$ is small. For the 5Hz, 7Hz, and 10Hz problems which have more nonzeros per column of B, besides decreasing the operation count more than the other strategies, `FT` exhibits equivalent or even better tree parallelism than `PO`. For such matrices, where $D(B) = nnz(B)/m$ is large, `FT` balances the work on the tree better than `PO` and reduces the work on the critical path more than the total work. Overall, `FT` reduces the operation count better than any other strategy and has good parallel properties.

5.2 Impact of the Blocking algorithm

First, we show that the Blocking algorithm decreases the operation count Δ while creating a limited number of groups. Second, we discuss and justify with parallelism arguments our clustering strategies illustrated in Figure 11.

In Table 5, we represent the value of $\frac{\Delta_{NG}}{\Delta_{min}}$ depending on the number of groups created. x means that the Blocking algorithm stopped because the condition $\Delta_{NG}/\Delta_{min} \leq \mu_0$ was reached, with μ_0 for Algorithm 3 set to 1.01. Computing from Table 5 the ratio of extra operations reduction $1 - \frac{\Delta_{NG}-\Delta_{min}}{\Delta_1-\Delta_{min}}$ for NG groups created, we observe an average reduction of 74% of the extra operations when $NG = 2$, *i.e.*, when only two groups are created. Table 5 also shows that Δ_{NG} reaches very quickly a value close to Δ_{min} and thus we confirm the expectation from Section 4.1 that RHS responsible for most extra operations were those intersecting a separator high in the tree.

Table 6: Sum of critical paths' operations ($\times 10^{13}$) for two grouping strategies when three groups are created.

$\sum_g \Delta_{cp}(g)$	5Hz	7Hz	10Hz	H0	H3
CritPathBuildGroup	**.092**	**.30**	**1.00**	**.037**	**.50**
RegBuildGroup	.12	.43	1.58	.044	.72

In Table 6, we report the sum of operation counts on the critical paths Δ_{cp} over all groups created using CritPathBuildGroup and RegBuildGroup strategies, when the number of groups created is three, leading to Δ close to Δ_{min}, see column "NG=3" of Table 5. In this case, the total number of operations Δ during the forward solution phase on all groups is equal whether we use CritPathBuildGroup or RegBuildGroup. Tree parallelism is thus a crucial discriminant between both strategies, and we indeed observe in Table 6 that CritPathBuildGroup effectively limits the length of critical paths over the three groups created, justifying its use.

5.3 Experiments with other orderings

As mentioned earlier, several orderings [14, 12, 2] may be used to order the unknowns of the original matrix, thanks to the algebraic nature of our Flat Tree and Blocking algorithms. Although local ordering methods (AMD, AMF as provided by the MUMPS package[3]) are known not to be competitive with respect to algebraic nested dissection-based approaches such as SCOTCH[4] or METIS[5] on large 3D problems, we include them in order to study how the Flat Tree and Blocking algorithms behave in general situations.

[3]http://mumps.enseeiht.fr/
[4]http://www.labri.fr/perso/pelegrin/scotch/
[5]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

Table 7: Operation count $\Delta(\times 10^{13})$ for permutation strategies `PO` and `FT`, and number of groups NG required to reach $\frac{\Delta_{NG}}{\Delta_{min}} \leq 1.01$ for blocking strategies `REG` and `BLK`. Different orderings (AMD, AMF, SCOTCH, METIS) are used.

orderings	AMD					AMF					SCOTCH					METIS				
σ	`PO`	`REG`	`FT`	`BLK`	Δ_{min}	`PO`	`REG`	`FT`	`BLK`	Δ_{min}	`PO`	`REG`	`FT`	`BLK`	Δ_{min}	`PO`	`REG`	`FT`	`BLK`	Δ_{min}
	Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG	
5Hz	1.44	53	**1.36**	4	1.25	**.75**	51	.87	7	.68	.47	328	**.32**	3	.25	.43	230	**.30**	3	.24
7Hz	5.03	38	**4.44**	4	4.35	**15.29**	18	17.82	12	14.89	1.60	287	**1.14**	3	.86	1.42	230	**1.08**	3	.82
10Hz	**19.34**	38	19.79	3	15.27	**96.86**	18	99.07	11	82.13	5.86	287	**4.21**	3	3.05	4.67	230	**3.44**	3	2.53
H0	.54	533	**.53**	4	.47	**.12**	333	.12	5	.0910	.0728	499	**.0627**	3	.0548	.0774	615	**.0668**	3	.0569
H3	134.54	380	**105.98**	5	9.07	**101.43**	63	133.97	17	9.26	2.19	615	**1.76**	5	1.18	1.95	533	**1.53**	5	1.12
H17	183.03	266	**225.66**	7	135.94	**467.06**	173	558.31	50	395.74	22.79	380	**18.97**	5	12.58	21.49	242	**16.80**	4	12.33
H116	2244.71	1	2244.71	1	2244.71	**39383.4**	1	39383.6	1	39383.4	290.45	109	**224.04**	4	153.16	263.72	78	**215.21**	4	157.00
S3	20.88	725	**17.85**	6	15.14	**20.71**	184	24.83	10	17.78	4.54	771	**3.41**	5	2.72	3.24	771	**2.64**	5	2.09
S21	392.57	685	**348.87**	5	310.63	**1141.45**	493	1352.32	77	830.51	50.91	492	**39.55**	4	31.73	34.31	223	**28.53**	5	24.86
S84	3025.30	352	**2847.53**	5	2501.38	**38664.7**	725	45346.4	213	30976.9	289.14	286	**228.31**	4	193.36	207.39	171	**174.43**	4	150.93
D30	**115.37**	111	121.29	8	94.51	**1015.16**	139	1279.55	75	825.21	16.72	156	**12.78**	5	8.77	15.52	144	**12.98**	5	8.61

First, an important aspect of using other orderings is that they often produce much more irregular trees, leading to a large number of pruned layers to sequence. The `FT` permutation reduces the operation count significantly with SCOTCH and METIS, for which we observe an average 31% and 26% reduction compared to the `PO` permutation. Gains are also obtained with AMD for most test cases. However, `FT` does not perform well with AMF. This can be explained by the fact that the former produces too irregular trees which do not fit well with the design of the `FT` strategy.

Second, we evaluate the Blocking algorithm (`BLK`) and compare it with a regular blocking algorithm (`REG`) based on the `PO` permutation, that divides the initial set of columns into regular chunks of columns. Table 7 shows that the number of groups required to reach $\frac{\Delta}{\Delta_{min}} \leq 1.01$ is much smaller for `BLK` than for `REG` in all cases. Our Blocking algorithm is very efficient with most orderings except AMF, where the number of groups created is high (but still lower than `REG`).

All preceding results confirmed that using the combination of the flat tree permutation and the Blocking algorithm, we are able to approach Δ_{min} within a sufficiently small tolerance. In the next section, we try to influence the ordering of the matrix to decrease Δ_{min}.

6 Guided Nested Dissection

At the moment, as soon as at least one RHS nonzero is present in a tree node, we considered in Section 1 that all operations involving the factors of that node are performed. A smaller granularity of sparsity (inner node sparsity) could be exploited by ordering last the indices of a supernode corresponding to RHS nonzeros. Because of fill-in, this is only useful for leaf nodes of the pruned tree $T_p(B)$. In general, those leaf nodes may not be very high in the tree, in which case there is not much gain to expect. Given an ordering and a tree, one may think of artificially moving the unknowns corresponding to RHS nonzeros to supernodes higher in the tree with on one side, a smaller pruned tree, but on the other side, an increase in the factor size due to these larger supernodes. Better, one may try to guide the nested dissection ordering in order to include as many nonzeros of B within separators during the top-down nested dissection and be able to prune larger subtrees. This will however involve a significant extra cost for applications where each RHS contains several contiguous nodes in the grid, *e.g.*, form a small parallelepiped. For such applications, the geometry of the RHS nonzeros could however be exploited. A first idea consists in avoiding problematic RHS by choosing separators that do not intersect RHS nonzeros. Although this idea could for example be tested by adding edges between RHS nonzeros before applying SCOTCH or METIS, this does not appear to be so useful in our case, where we observed much overlap between successive RHS. Another idea, when all RHS are localized in a specific area of the domain, consists in shifting the separators from the nested dissection to insulate the RHS in a small part of the domain. Such a modification of the ordering yields an unbalanced tree in which the RHS nonzeros appear at the smaller side of the tree, improving the efficiency of tree pruning and resulting in a reduction of Δ_{min}, and thus Δ. This so called *guided* nested dissection was implemented and tested on the set of test cases shown in Table 8, where we observe that the number of operations Δ_{min} is decreased, as expected. Since the factor size has also increased significantly, one may need to find a trade-off in order to avoid increasing too much the cost of the factorization.

Table 8: Number of operations Δ_{min} ($\times 10^{13}$) and factor size ($\times 10^{9}$) for the original (ND) and for the guided (GND) nested dissection orderings.

Matrices	5Hz		7Hz		10Hz		H0		H3	
Strategy	ND	GND	ND	GND	ND	GND	ND	GND	ND	GND
Δ_{min}	.22	**.19**	.69	**.62**	2.26	**1.99**	.050	**.025**	.95	**.81**
factor size	**3.72**	5.18	**12.8**	19.7	**44.8**	73.4	**.24**	.37	**4.50**	5.57

7 Applications and related problems

We illustrate the scope of this work by presenting applications where our contributions can be applied. In applications requiring only part of the solution, one can show that the tools presented in Section 2 can be applied to the backward substitution ($UX = Y$), which involves similar mechanisms as the forward elimination [17, 19]. The backward substitution traverses the tree nodes from top to bottom so that the interval mechanism is reversed, *i.e.*, the interval from a parent node includes the intervals from its children and the properties of local optimality are preserved. If the structure of the partial solution requested differs from the RHS structure, another call to the Flat Tree algorithm must then be performed to optimize the number of operations. Exploiting sparsity also in the backward step can for instance be useful in some augmented approaches [20] to deal with small matrix updates without complete refactoring, and in some 3D EM geophysics applications [16]. Another application of this work is the computation of Schur complements, where instead of truncating a factorization of the whole system $\left(\begin{smallmatrix} A & C \\ B & D \end{smallmatrix}\right)$, one exploits the factorization of A to use triangular solves with sparse RHS. Taking the symmetric case where $C = B^T$, the Schur complement S can be written $S = D - BA^{-1}B^T = D - B(LL^T)^{-1}B^T = D - (L^{-1}B^T)^T(L^{-1}B^T)$, as in the PDSLin solver [18]. Since B is sparse, $B' = L^{-1}B^T$ can be computed thanks to the algorithms developed in this article before computing the sparse product $B'^T B'$.

To conclude, we comment on related problems and algorithms. We have seen that the Blocking algorithm is closely related to graph algorithms like coloring and maximum independent set. Concerning the minimization problem (10) which we addressed with the Flat Tree algorithm, it can also be regarded globally: using the structure of $L^{-1}B$, the problem then consists in finding a permutation of the columns that minimizes the sum of the intervals weighted with δ_u. This interval minimization problem is similar to a sparse matrix profile reduction problem [5, 15]. As mentioned in Section 4.1, hypergraph models have been used in the context of blocking algorithms, with different constraints and objectives compared to ours [3, 18]. Modeling $L^{-1}B$ as an hypergraph might lead to other heuristics than the Flat Tree algorithm using some variants of hypergraph partitioning, although dense parts in $L^{-1}B$ might need special treatment. One advantage of our permutation and blocking algorithms is that, instead of tackling the problem globally, they decompose the problem into easier subproblems with low complexity by making use of the separator tree T, thereby exploiting the fact that $L^{-1}B$ has a very special structure closely related to the tree.

Conclusion

Table 9: Time (s) of the forward elimination according to the strategy used on a single Intel Xeon core @2.3GHz.

Times	DEN	TP	INI	PO	TP	BLK
H0	881.9	156.2	120.8	95.7	78.1	65.4
5Hz	1527.6	472.3	274.3	224.1	180.0	138.6

We introduced permutation and blocking algorithms to further improve the tree pruning [11, 17] and the node interval [4] algorithms introduced in previous work. A first main contribution of this article is to provide a "flat tree" algorithm to permute right-hand sides in order to reduce the cost of the forward elimination. As a second contribution, we introduced a Blocking algorithm that further decreases this cost by adequately choosing groups of right-hand sides that can be processed together. Although both algorithms are based on geometrical observations, they are designed with an algebraic approach, giving a general scope to this work. Notions of node optimality and RHS independence were introduced and formalized, together with theoretical properties to provide insight and to support the proposed algorithms. Experimental results on real test cases confirmed the effectiveness of both the *Flat Tree* and the *Blocking* algorithms. Compared to a Postorder-based permutation, the Flat Tree permutation showed an average (resp. maximum) gain of 13% (resp. 25%) on the total operation count with a nested dissection ordering, and interesting parallel properties. Moreover, results with the Blocking algorithm validate our approach since only a handful of groups is created compared to several hundreds when using a regular blocking technique. Finally, Table 9 shows that in a sequential setting, time reduction follows operation reduction on the smallest of our two sets of problems. A detailed performance analysis in multithreaded and distributed environments is out of the scope of this study and will be the object of future work.

Acknowledgements

We thank EMGS and Seiscope for providing the test cases, and F.-H. Rouet for his comments on a previous version of this paper. This work was partially supported by the MUMPS Consortium and by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

References

[1] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto, *Fast 3D frequency-domain full waveform inversion with a*

parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea, Geophysics, 81 (2016), pp. R363 – R383.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software, 33(3) (2004), pp. 381–388.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar, *On computing inverse entries of a sparse matrix in an out-of-core environment*, SIAM Journal on Scientific Computing, 34 (2012), pp. A1975–A1999.

[4] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and F.-H. Rouet, *Parallel computation of entries of A^{-1}*, SIAM Journal on Scientific Computing, 37 (2015), pp. C268–C284.

[5] M. W. Berry, B. Hendrickson, and P. Raghavan, *Sparse marix reordering schemes for browsing hypertext*, Lecture notes in applied mathematic, 32 (1996), pp. 99–124.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[7] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, London, 1986.

[8] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[9] J. A. George, *Nested dissection of a regular finite-element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

[10] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM Journal on Matrix Analysis and Applications, 15 (1994), pp. 62–79.

[11] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM Journal on Matrix Analysis and Applications, 14 (1993), pp. 334–352.

[12] G. Karypis and K. Schloegel, *ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 4.0*, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., Aug. 2003. Users' manual.

[13] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[14] F. Pellegrini, Scotch *and* libscotch 5.0 *User's guide*, Technical Report, LaBRI, Université Bordeaux I, 2007.

[15] J. K. Reid and J. A. Scott, *Reducing the total bandwidth of a sparse unsymmetric matrix*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 805–821.

[16] D. V. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, *Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver*, Geophysical Journal International, 209 (2017), pp. 1558–1571.

[17] Tz. Slavova, *Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems*, Ph.D. dissertation, Institut National Polytechnique de Toulouse, Apr. 2009.

[18] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *On partitioning and reordering problems in a hierarchically parallel hybrid linear solver*, in 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Cambridge, MA, United States, 2013, pp. 1391–1400.

[19] Y.-H. Yeung, J. Crouch, and A. Pothen, *Interactively cutting and constraining vertices in meshes using augmented matrices*, ACM Trans. Graph., 35 (2016), pp. 18:1–18:17.

[20] Y. H. Yeung, A. Pothen, M. Halappanavar, and Z. Huang, *AMPS: an augmented matrix formulation for principal submatrix updates with application to power grids*, SIAM Journal on Scientific Computing, 39 (2017), pp. S809–S827.

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399